	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorio de Computacion Salas A y B

Profesor(a): Davila Perez Rene Adrian

Asignatura: Programacion orientada a objetos

Grupo: 01

No de practica(s): Proyecto 3

Integrante(s): 322001549, 322006829, 322125533, 425131152, 322103351

No de lista o brigada: 07

Semestre: 2026-1

Fecha de entrega: Lunes 01 de diciembre 2025

Observaciones:

Calificacion:

Índice

1. Introducción

El desarrollo de aplicaciones móviles se ha convertido en una de las áreas más dinámicas y relevantes dentro de la ingeniería de software. A lo largo del curso se han revisado conceptos fundamentales de programación, diseño de interfaces y lógica de sistemas, utilizando Flutter como marco de trabajo. Como proyecto integrador, se plantea la creación de una aplicación que simule el sistema de batallas del videojuego Pokémon, con el fin de aplicar de manera práctica los conocimientos adquiridos y demostrar la capacidad de diseñar un sistema interactivo, modular y funcional.

1.1. Planteamiento del problema

Los videojuegos de rol y estrategia, como Pokémon, se caracterizan por sistemas de combate basados en turnos, donde las decisiones del jugador y las características de los personajes determinan el resultado de la batalla. Replicar este tipo de mecánicas en una aplicación móvil implica enfrentar diversos retos:

- Implementar un sistema de turnos que respete las reglas de prioridad y finalización de la batalla.
- Representar las ventajas y desventajas de los tipos de ataque, siguiendo la lógica de daño doble, mitad o nulo.
- Diseñar un menú interactivo que muestre al Pokémon del usuario, al rival y las opciones de ataque disponibles.

1.2. Motivación

La motivación principal de este proyecto es aplicar de manera práctica los conceptos aprendidos en el curso y demostrar cómo Flutter puede ser utilizado para desarrollar aplicaciones interactivas más allá de las tradicionales interfaces de usuario. Recrear un sistema de batallas Pokémon no solo resulta atractivo por su popularidad y valor lúdico, sino que también representa un desafío técnico que fomenta:

- El pensamiento lógico y la correcta estructuración de algoritmos.
- La La modularidad y reutilización de código, al definir clases y métodos que representen Pokémon, ataques y estados.
- El trabajo colaborativo, ya que cada miembro del equipo desarrolla una parte específica del proyecto (objetivos, diagramas UML, desarrollo paso a paso, resultados).
- La creatividad y motivación personal, al integrar elementos adicionales como música y estados alterados que enriquecen la simulación.

1.3. Objetivos

- Implementar atributos básicos de los Pokémon: vida (HP) y velocidad.
- Diseñar un sistema de turnos que determine el orden de ataque según la velocidad.
- Definir al menos un Pokémon y un ataque de cada tipo, respetando las reglas de efectividad (doble, mitad, nulo).
- Construir un menú interactivo que muestre los Pokémon en combate y las opciones de ataque.

2. Conceptos

En este apartado se explicaran los conceptos o las herramientas que fueron utilizadas y como funcionan.

2.1. Clases y objetos

Dart es un lenguaje orientado a objetos con clases y herencia basada en mixins. Cada objeto es una instancia de una clase, y todas las clases, excepto Null, descienden de Object. La herencia basada en mixins significa que, aunque cada clase (excepto la clase superior, ¿Object?) tiene exactamente una superclase, el cuerpo de una clase se puede reutilizar en múltiples jerarquías de clases. Los métodos de extensión son una forma de añadir funcionalidad a una clase sin cambiarla ni crear una subclase. Los modificadores de clase permiten controlar cómo las bibliotecas pueden subtipificar una clase.[1]

En la programación orientada a objetos , un objeto es una unidad autocontenida de código y datos. Los objetos se crean a partir de plantillas llamadas clases. Un objeto se compone de propiedades (variables) y métodos (funciones). Un objeto es una instancia de una clase.[1]

- Utilizando miembros de clase:

  Los objetos tienen miembros que consisten en funciones y datos (métodos y variables de instancia , respectivamente). Al llamar a un método, se invoca sobre un objeto: el método tiene acceso a las funciones y datos de ese objeto. Utilice un punto (.) para referirse a una variable de instancia o método:

```
var p = Point(2, 2);

// Get the value of y.
assert(p.y == 2);

// Invoke distanceTo() on p.
double distance = p.distanceTo(Point(4, 4));
```

  Utilice ?.en su lugar .para evitar una excepción cuando el operando más a la izquierda sea nulo:

```
// If p is non-null, set a variable equal to its y value.
var a = p?.y;
```

- Utilizando constructores Puedes crear un objeto usando un constructor . Los nombres de los constructores pueden ser ‘init‘ ClassNameo ‘init‘ . Por ejemplo, el siguiente código crea objetos usando los constructores ‘init‘ y ‘init‘: ClassName.identifierPointPoint()Point.fromJson().[1]

```
var p1 = Point(2, 2);
var p2 = Point.fromJson({'x': 1, 'y': 2});
```

  El siguiente código tiene el mismo efecto, pero utiliza la newpalabra clave opcional antes del nombre del constructor:

```
var p1 = new Point(2, 2);
var p2 = new Point.fromJson({'x': 1, 'y': 2});
```

- Variables y métodos de clase

  utilice la static palabra clave para implementar variables y métodos de toda la clase.

```
class Queue {
  static const initialCapacity = 16;

}

void main() {
  assert(Queue.initialCapacity == 16);
}
```

- métodos estáticos

  Los métodos estáticos (métodos de clase) no operan sobre una instancia y, por lo tanto, no tienen acceso a ella this. Sin embargo, sí tienen acceso a las variables estáticas. Como muestra el siguiente ejemplo, los métodos estáticos se invocan directamente en una clase.[1]

```
import 'dart:math';

class Point {
  double x, y;
  Point(this.x, this.y);

  static double distanceBetween(Point a, Point b) {
    var dx = a.x - b.x;
    var dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
  }
}

void main() {
  var a = Point(2, 2);
  var b = Point(4, 4);
  var distance = Point.distanceBetween(a, b);
  assert(2.8 < distance && distance < 2.9);
  print(distance);
}
```

2.2. Constructores

Los constructores son funciones especiales que crean instancias de clases.

Dart implementa muchos tipos de constructores. Excepto los constructores por defecto, estas funciones utilizan el mismo nombre que su clase.[1]

1. constructores generativos Para instanciar una clase, utilice un constructor generativo.

```
class Point {
  // Instance variables to hold the coordinates of the point.
  double x;
```

```
  double y;

  // Generative constructor with initializing formal parameters:
  Point(this.x, this.y);
}
```

2. Constructores por defecto

   Si no declaras un constructor, Dart utiliza el constructor predeterminado. El constructor predeterminado es un constructor generativo sin argumentos ni nombre.[1]

3. Constructores con nombre

   Utilice un constructor con nombre para implementar múltiples constructores para una clase o para proporcionar mayor claridad:

```
vconst double xOrigin = 0;
const double yOrigin = 0;

class Point {
  final double x;
  final double y;

  // Sets the x and y instance variables
  // before the constructor body runs.
  Point(this.x, this.y);

  // Named constructor
  Point.origin() : x = xOrigin, y = yOrigin;
}
```

   Una subclase no hereda el constructor con nombre de la superclase. Para crear una subclase con un constructor con nombre definido en la superclase, implemente dicho constructor en la subclase.

4. Constructores constantes

   Si tu clase produce objetos que no cambian, haz que estos objetos sean constantes en tiempo de compilación. Para ello, define un constconstructor con todas las variables de instancia establecidas como final.[1]

```
class ImmutablePoint {
  static const ImmutablePoint origin = ImmutablePoint(0, 0);

  final double x, y;

  const ImmutablePoint(this.x, this.y);
}
```

   Los constructores constantes no siempre crean constantes. Pueden invocarse fuera de un constcontexto. Para obtener más información, consulte la sección sobre el uso de constructores .

5. Constructores de redireccionamiento

   Un constructor puede redirigir a otro constructor de la misma clase. Un constructor que redirige tiene un cuerpo vacío. El constructor utiliza thisen lugar del nombre de la clase después de dos puntos (:).

```
class Point {
  double x, y;

  // The main constructor for this class.
  Point(this.x, this.y);

  // Delegates to the main constructor.
  Point.alongXAxis(double x) : this(x, 0);
}
```

2.3. Encapsulacion, Getters y Setters

En Dart, la encapsulación consiste en ocultar datos dentro de una biblioteca, protegiéndolos de factores externos. Esto ayuda a controlar el programa y evita que se vuelva demasiado complejo.[2]

La encapsulación se puede lograr mediante:

- Declarar las propiedades de la clase como privadas utilizando guion bajo () .
- Proporcionar métodos públicos getter y setter para acceder y actualizar el valor de la propiedad privada.

Nota: Dart no admite palabras clave como 'public' , 'private' y 'protected. Dart utiliza "'_" (guion bajo) para indicar que una propiedad o método es privado. Los métodos getter y setter se utilizan para acceder y actualizar el valor de una propiedad privada. Los métodos getter se utilizan para acceder al valor de una propiedad privada. Los métodos setter se utilizan para actualizar el valor de una propiedad privada.[2]

Ejemplo 1: En este ejemplo, crearemos una clase llamada `Employee`. Esta clase tendrá dos propiedades privadas: _id y _name. También crearemos dos métodos públicos, `getId()` y `getName()`, para acceder a las propiedades privadas. Asimismo, crearemos dos métodos públicos, `setId()` y `setName()`, para actualizar las propiedades privadas.[2]

```
class Employee {
  // Private properties
  int? _id;
  String? _name;

// Getter method to access private property _id
  int getId() {
    return _id!;
  }
// Getter method to access private property _name
  String getName() {
    return _name!;
  }
// Setter method to update private property _id
```

```
15    void setId(int id) {
16      this._id = id;
17    }
18  // Setter method to update private property _name
19    void setName(String name) {
20      this._name = name;
21    }
22
23  }
24
25  void main() {
26    // Create an object of Employee class
27    Employee emp = new Employee();
28    // setting values to the object using setter
29    emp.setId(1);
30    emp.setName("John");
31
32    // Retrieve the values of the object using getter
33    print("Id: ${emp.getId()}");
34    print("Name: ${emp.getName()}");
35  }
```

En Dart, puedes controlar el acceso a las propiedades e implementar la encapsulación mediante el uso de propiedades de solo lectura. Para ello, añade la palabra clave ‘ final‘ antes de la declaración de la propiedad. De esta forma, solo podrás acceder a su valor, pero no modificarlo.[2]

```
1  class Student {
2    final _schoolname = "ABC School";
3
4    String getSchoolName() {
5      return _schoolname;
6    }
7  }
8
9  void main() {
10   var student = Student();
11   print(student.getSchoolName());
12   // This is not possible
13   //student._schoolname = "XYZ School";
14 }
```

Puedes crear métodos getter y setter usando las palabras clave `get` y `set`. En el siguiente ejemplo, hemos creado una clase llamada `Vehicle`. Esta clase tiene dos propiedades privadas: _model y _year. También hemos creado dos métodos getter y setter para cada propiedad. Estos métodos se llaman `model` y `year`, y se utilizan para acceder y actualizar el valor de las propiedades privadas.[2]

```
1  class Vehicle {
2    String _model;
3    int _year;
```

```
4
5     // Getter method
6     String get model => _model;
7
8     // Setter method
9     set model(String model) => _model = model;
10
11    // Getter method
12    int get year => _year;
13
14    // Setter method
15    set year(int year) => _year = year;
16  }
17
18  void main() {
19    var vehicle = Vehicle();
20    vehicle.model = "Toyota";
21    vehicle.year = 2019;
22    print(vehicle.model);
23    print(vehicle.year);
24  }
```

¿Por qué es importante la encapsulación?

1. Ocultación de datos : La encapsulación oculta los datos del exterior. Impide que el código fuera de la clase acceda a ellos. Esto se conoce como ocultación de datos.[2]
2. Facilidad de prueba : La encapsulación permite probar la clase de forma aislada. Esto permite probar la clase sin necesidad de probar el código fuera de ella.[2]
3. Flexibilidad : La encapsulación permite cambiar la implementación de la clase sin afectar al código que se encuentra fuera de la clase.[2]
4. Seguridad : La encapsulación permite restringir el acceso a los miembros de la clase. Esto permite limitar el acceso a los miembros de la clase desde el código externo a la biblioteca.[2]

con todo lo anterior la aplicacion de estos conceptos para el proyecto en cuestion se observa de la siguiente manera:

```
1   class Pokemon {
2     String nombre;
3     TipoPokemon tipo;
4     int hp;
5     int velocidad;
6     List<Ataque> ataques;
7
8     String imagePath;
9
10  }
```

2.4. Enum

En Dart, los enums extienden de la clase Enum, y nos sirven para agrupar conjuntos de valores como veremos más adelante.[3]

para declarar una enumeración simple, debemos usar la palabra clave enum el nombre debe empezar con mayúscula y allí seguido una lista de valores a enumerar.[3]

```
enum Color { red, greeen, blue }
enum DayOfWeek {
  monday,
  tuesday,
  wednesday,
  thursday,
  friday,
  saturday,
  sunday
}
```

Dart también permite en las clases de enums declarar atributos, funciones y constructores.

Para declarar enums mejorados, se usa de la misma forma que las clases normales, con algunas particularidades:

- las variables o atributos deben ser final.
- Todos los constructores deben ser constantes.
- Enum se extiende automáticamente.
- No se pueden crear variables con el nombre values porque entra en conflicto con la propiedad de los enums.
- Todos los casos de enums deben ser declarados al principio.

en el proyecto:

```
enum TipoPokemon {
  FUEGO,
  AGUA,
  PLANTA,
  ELECTRICO,
  ROCA,
  NORMAL,
  LUCHA,
  VENENO,
  TIERRA,
  VOLADOR,
  PSIQUICO,
  BICHO,
  FANTASMA,
  DRAGON,
  DARK,
  STEEL
}
```

```
var lanzallamas =
Ataque(nombre: 'Lanzallamas', tipo: TipoPokemon.FUEGO, poder: 90);

var placaje =
Ataque(nombre: 'Placaje', tipo: TipoPokemon.NORMAL, poder: 40);
var pistolaAgua =
Ataque(nombre: 'Pistola Agua', tipo: TipoPokemon.AGUA, poder: 40);
```

2.5. Colecciones

En un mapa, cada elemento es un par clave-valor. Cada clave dentro de un par está asociada a un valor, y tanto las claves como los valores pueden ser de cualquier tipo de objeto. Cada clave solo puede aparecer una vez, aunque el mismo valor puede estar asociado a varias claves diferentes. La compatibilidad de Dart con los mapas se proporciona mediante literales de mapa y el Map tipo.[4]

para el proyecto:

```
const Map<TipoPokemon, Map<TipoPokemon, double>> mapaDeTipos =
```

Quizás la colección más común en casi todos los lenguajes de programación sea el array , o grupo ordenado de objetos. En Dart, los arrays son Listobjetos, por lo que la mayoría de la gente los llama simplemente listas .[4]

Las literales de lista de Dart se representan mediante una lista de elementos separados por comas y encerrados entre corchetes ([]). Cada elemento suele ser una expresión. A continuación, se muestra una lista de Dart sencilla:

Las listas utilizan indexación basada en cero, donde 0 es el índice del primer elemento y 0 list.length - 1es el índice del último. Puede obtener la longitud de una lista mediante la .lengthpropiedad y acceder a sus elementos mediante el operador de subíndice ([]):

```
List<String> battleLog = ['¡La batalla ha comenzado!'];
```

2.6. Widgets

La Interfaz de Usuario en Flutter, está compuesta de Widgets. Los Widgets en Flutter se construyen utilizando un Framework moderno inspirado en React. Casi todo en Flutter está compuesto de Widgets. Los Widgets describen como sería su vista dado una configuración y estado actual. Cuando el estado cambia, el Widget se reconstruye.[5]

Los Widgets son subclases de StatelessWidget o StatefulWidget. Todo depende de si se quiere guardar el estado o no del widget. Todos los widgets deben implementar la función build(), dentro de este se va armando la estructura necesaria para el widget requerido. El StatelessWidget se utiliza cuando no es necesario conservar el estado del widget, para guardar el estado o alguna otra característica de un widget es recomendable usar StatefulWidget o algún manejador de estados.[5]

Widgets básicos

- Text: Es un widget que permite crear textos con estilos personalizados.
- Row: Es un widget que permite tener un diseño flexible y tener widgets hijos en un conjunto horizontal (Fila).

- Column: Es un widget al igual que el Row, es de diseño flexible y permite tener widgets hijos en un conjunto vertical (Columna).
- Stack: El widget Stack permite colocar widgets uno encima de otro. Es un widget que posiciona a sus hijos en relación con los bordes de su caja. Aquí se puede utilizar otro widget Positioned que se encarga de controlar en dónde se van a ubicar los hijos de Stack.
- Container: El widget Container permite crear un elemento rectangular, el cual se puede decorar con estilos, el fondo, sombras y formas con otro widget BoxDecoration. El widget BoxDecoration básicamente es un widget que nos permite pintar o dibujar una caja en blanco a nuestra necesidad.

En el proyecto

```
return Scaffold(
      appBar: AppBar(
        title: const Text('Batalla Pokémon'),
      ),
```

```
Widget _buildAttackMenu() {

    //Comprueba si la batalla termina
    bool batallaTerminada = (pokemonUsuario.hp == 0 || pokemonRival.hp == 0);
    //Obtiene los ataques de nuestro Pokémon
    List<Ataque> ataques = pokemonUsuario.ataques;

    //Crea una lista de Widgets (botones)
    List<Widget> botonesDeAtaque = [];

    for (var ataque in ataques) {
      botonesDeAtaque.add(
        Padding(
          padding: const EdgeInsets.symmetric(vertical: 4.0),
          child: ElevatedButton(
            //Si la batalla terminó, onPressed es null
            onPressed: batallaTerminada
                ? null
                : () {
                    _atacar(ataque);
                  },
            child: Text('${ataque.nombre} (${ataque.tipo.name})'),
          ),
        ),
      );
    }
```

2.7. Polimorfismo

Poli significa muchos y morfo significa formas . El polimorfismo es la capacidad de un objeto para adoptar muchas formas. Como seres humanos, tenemos la capacidad de adoptar muchas formas. Podemos ser estudiantes, profesores, padres, amigos, etc. De manera similar, en la programación orientada a objetos, el polimorfismo es la capacidad de un objeto para adoptar muchas formas.[2]

La sobreescritura de métodos es una técnica que permite crear un método en la clase hija con el mismo nombre que el método en la clase padre. El método en la clase hija sobreescribe el método en la clase padre.[2]

```
class ParentClass{
void functionName(){
  }
}
class ChildClass extends ParentClass{
@override
void functionName(){
  }
}
```

Ejemplo 1: Polimorfismo mediante la sobreescritura de métodos en Dart

En el siguiente ejemplo, existe una clase llamada Animal con un método llamado eat() . El método eat() se redefine en la clase hija llamada Dog .

```
class Animal {
  void eat() {
    print("Animal is eating");
  }
}

class Dog extends Animal {
  @override
  void eat() {
    print("Dog is eating");
  }
}

void main() {
  Animal animal = Animal();
  animal.eat();

  Dog dog = Dog();
  dog.eat();
}
```

Ventaja de usar polimorfismo:

1. Las subclases pueden anular el comportamiento de la clase padre.
2. Nos permite escribir código más flexible y reutilizable.

2.8. Null Safety

La seguridad nula evita errores que resultan del acceso no intencional a las variables establecidas en null. Por ejemplo, si un método espera un entero pero recibe null, la aplicación genera un error de ejecución. Este tipo de error, un error de desreferencia nula, puede ser difícil de depurar.

Con seguridad nula sólida, todas las variables requieren un valor. Esto significa que Dart considera que todas las variables no aceptan valores nulos . Solo se pueden asignar valores del tipo declarado, como int i=42. Nunca se puede asignar un valor de nulla los tipos de variable predeterminados. Para especificar que un tipo de variable puede tener un nullvalor, agregue un ? después de la anotación de tipo: int? i. Estos tipos específicos pueden contener un valor de null o un valor del tipo definido.

La seguridad de nulos convierte los posibles errores de ejecución en errores de análisis en tiempo de edición . Con la seguridad de nulos, el analizador y los compiladores de Dart detectan si una variable no nula presenta alguna de las siguientes características:

No se ha inicializado con un valor distinto de nulo Se le ha asignado un nullvalor.

Dart admite la seguridad nula utilizando los siguientes dos principios de diseño básicos:

- No nulo por defecto: A menos que se indique explícitamente a Dart que una variable puede ser nula, se considera no nula. Esta opción predeterminada se eligió después de que una investigación revelara que la opción no nula era, con diferencia, la más común en las API.
- Totalmente sano: La seguridad nula de Dart es sólida. Si el sistema de tipos determina que una variable o expresión tiene un tipo no nulo, se garantiza que nunca podrá evaluarse nullen tiempo de ejecución.

En el codigo:

```
1  late Pokemon pokemonUsuario;
2   late Pokemon pokemonRival;
3   late Batalla batalla;
```

Con setters que aceptan solo valores no nulos:

```
1 pokemonUsuario = Pokemon(
2       nombre: 'Charizard',
3       tipo: TipoPokemon.FUEGO,
4       hp: 78,
5       velocidad: 100,
6       ataques: [lanzallamas, placaje],
7       //ruta de pubspec.yaml
8       imagePath: 'assets/imagenes/charizard.png',
9     );
```

3. Desarrollo

3.1. Lógica de combate, clases y abstracción

Se definieron modelos modelos de datos para representar las entidades del juego creando la clase *Pokemon* para abstraer atributos como salud, velocidad, tipo elemental y ruta de imagen. De igual forma, se modelaron las clases *Ataque* e *Item* para gestionar las acciones ofensivas y defensivas respectivamente. Su lógica crítica del combate se aisló dentro de la clase batalla, encapsulando las reglas de combate tales como, la determinación del orden de turnos basados en su velocidad o bien el cálculo del daño.

La clase *BattleScreen* extiende a *StateFulWidget*, lo que permite el uso de un estado mutable para actualizar de forma dinámica la interfaz (barras de vida, bitácora de combate) cada vez que ocurría una interacción, utilizando el método *setState* para re dibujar los widgets necesarios sin recargar toda la aplicación. Respecto a las fortalezas y debilidades, fue posible hacer una relación de los tipos de atacantes con los defensores mediante una estructura de datos tipo mapa (diccionario), al realizar dicha relación se devuelven los multiplicadores de daño en *const Map*.

La simulación comienza con la clase *Batalla*, razón por las que esta y las clases anteriores decidimos separarlas de la interfaz gráfica. La clase determina el orden de ataque comparando el atributo *velocidad* de los objetos *Pokemon* involucrados, utilizando así un encapsulamiento mediante la variable privada *_atacanteActual* para controlar el flujo.

A través de *realizarAtaque* implementa la lógica matemática del combate, utiliza una colección tipo *Map* para verificar la relación entre el tipo del ataque y el tipo del defensor, aplicando los multiplicadores correspondientes según el pokémon, lo anterior culmina en una lista de cadenas de texto que describe el resultado del turno. Adicionalmente, existe *usarItem* para gestionar una lógica defensiva invocando al método *curar* del pokémon y consumir su turno.

3.2. Interfaz Gráfica

Para que exista una interacción visual, se implementa la clase *_BattleScreenState* que extiende a *State* para mejorar el ciclo de vida de la aplicación, para comenzar con el combate, la función *_inicializarBatalla* crea los pokemon con sus estadísticas y la batalla. Adicionalmente las funciones *_scrollToBottom* para un ajuste automático del log durante el combate, así como *_abrirMochila* y *_usarObjeto* para la curación del jugador.

La interfaz de usuario se desarrolló siguiendo el paradigma declarativo de Flutter, donde la interfaz es un reflejo del estado actual de la aplicación con una arquitectura de Composición de Widgets para construir una interfaz compleja a partir de elementos simples.

En la arquitectura de capas, se coloca una capa de fondo que funciona como un contenedor que renderiza la imagen del estadio *AssetImage* ajustada para cubrir la totalidad del dispositivo. De forma invisible, también existe una capa que es el área central donde se posicionan los personajes mediante coordenadas relativas *Align*, permitiendo que el diseño se adapte a diferentes tamaños de pantalla (responsividad). Debajo del área del combate también se encuentra una capa superior que contiene la bitácora del texto para que siempre sean visibles.

Las imágenes son manipuladas geométricamente con *Transform.scale* con un factor negativo en el eje X para invertir horizontalmente la imagen del usuario (Charizard), logrando que ambos combatientes se miren frente a frente sin necesidad de editar los archivos de imagen originales. Los personajes son acompañados con sus respectivas barras de vida, estas cuentan con widgets personalizados utilizando un *LinearProgressIndicator* cuyo color cambia dinámicamente (Verde,

Amarillo, Rojo) mediante una evaluación condicional del porcentaje de vida restante (hp / maxHp), proporcionando retroalimentación visual inmediata al jugador.

Gestión del estado

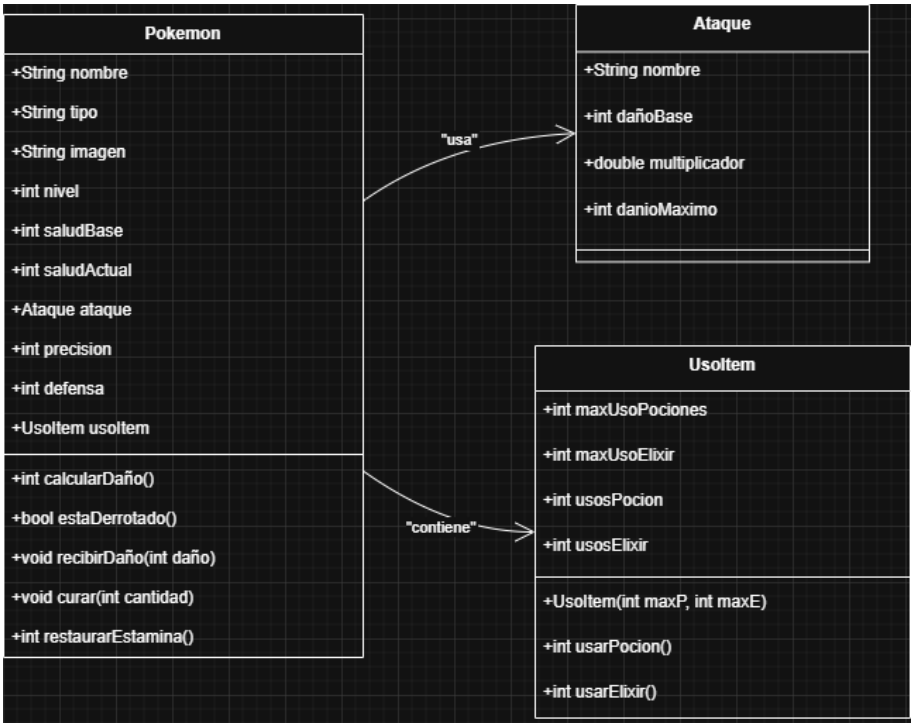
La interactividad se gestiona mediante la clase $_{B}attleScreenState$. Funciones asíncronas (_atacar, _usarObjeto) invocan al método *setState*, el cual notifica al framework sobre cambios en los datos, provocando la reconstrucción eficiente de solo los widgets afectados (como la reducción de la barra de vida o la aparición del botón de reinicio).

Gestión dinámica de controles

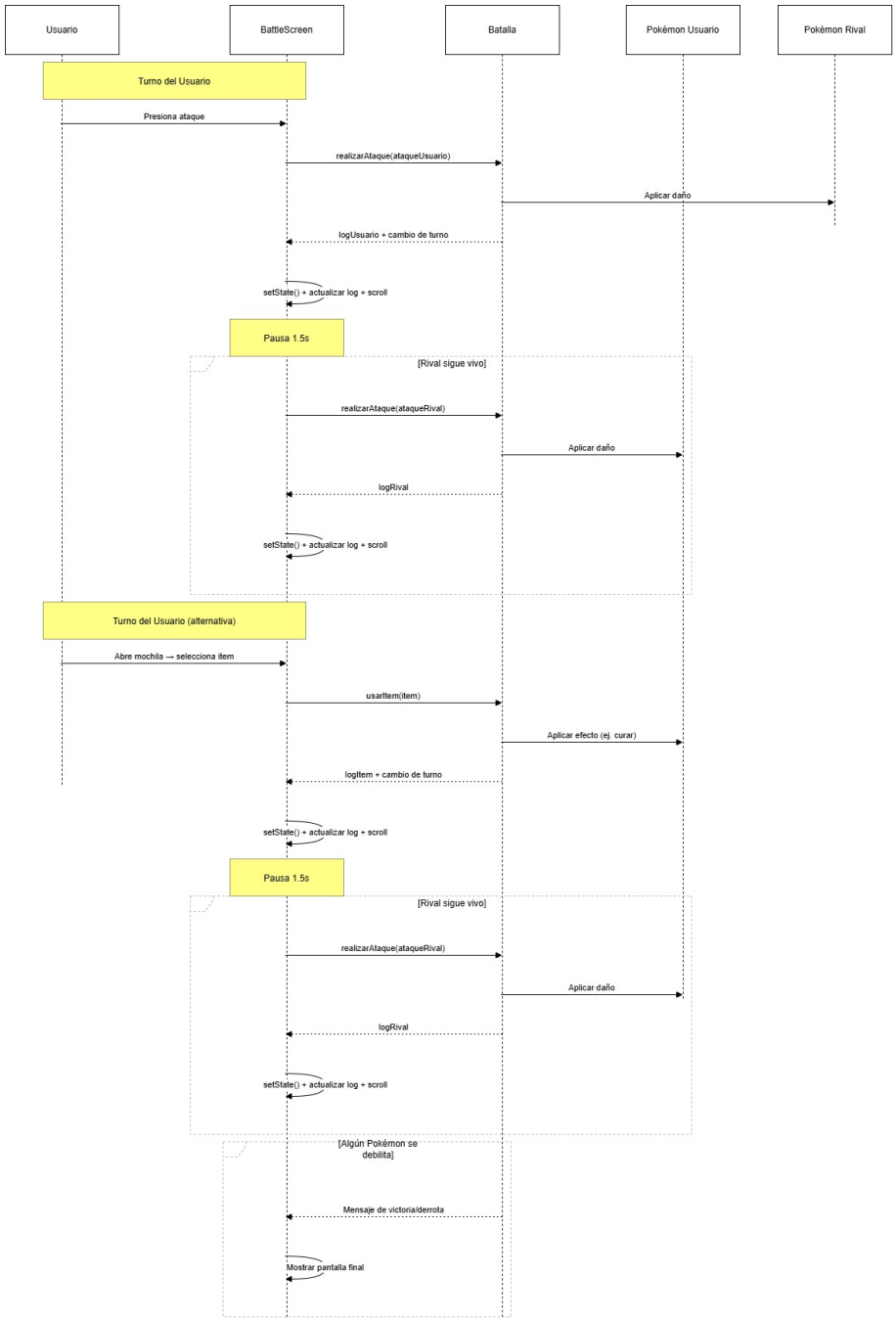
En lugar de codificar botones estáticos, se implementó una función generadora que itera sobre la colección de ataques del objeto Pokemon del usuario $\_buildAttackMenu$. Esta función construye dinámicamente una columna de botones *ElevatedButton*, asignando a cada uno el nombre y tipo del ataque correspondiente. Adicionalmente, se implementó una lógica de validación de estado en la propiedad *onPressed*: si la salud de algún combatiente llega a cero, los botones se deshabilitan automáticamente (null), impidiendo acciones fuera de turno o tras finalizar la partida.

Lo anterior va directamente relacionado con el controlador de la secuencia de combate *_atacar* que refleja todo el flujo de un turno de batalla, invoca al método $realizarAtaque$ de la clase *Batalla*. Inmediatamente utiliza *setState* para reflejar el daño en la barra de vida del rival y agregar los mensajes resultantes a la bitácora visual, posteriormente, dispara la función de auto-scroll para asegurar que el jugador visualice el resultado de su acción. Finalmente utiliza un $Future.delayed$ para introducir una pausa artificial de 1.5 segundos. Tras este lapso, ejecuta automáticamente el turno del rival invocando nuevamente la lógica de ataque y actualización de la interfaz, completando así el ciclo del turno sin bloquear el hilo principal de la aplicación.

3.3. Diagrama UML estático



3.4. Diagrama UML Dinamico



4. Resultados

Durante el desarrollo de la aplicación se obtuvo un sistema de batalla completamente funcional que simula las mecánicas básicas del videojuego Pokémon. En primer lugar, se implemento una estructura interna capaz de gestionar todos los elementos fundamentales del combate: cada Pokémon cuenta con puntos de vida (HP), velocidad, tipo elemental y una lista de ataques con daños y características específicas. Con base en estos elementos, la aplicación decide automáticamente cuál Pokémon actúa primero, siempre otorgando prioridad al que posea mayor velocidad. Este comportamiento se observa de manera consistente en las pruebas realizadas.

A lo largo de la implementación, también se integró la tabla de ventajas y desventajas de tipos. Los resultados muestran que los ataques infligen daño doble, normal, reducido o nulo según la relación de tipos entre los Pokémon combatientes. Esto permitió reproducir fielmente el comportamiento esperado del sistema original. Cada acción altera visualmente los puntos de vida del rival, lo cual puede apreciarse en las capturas insertadas a continuación.

Figura 1: Actualización de HP

La pantalla principal del combate despliega correctamente al Pokémon del usuario y al Pokémon rival, junto con sus estadísticas básicas. Además, el menú de batalla permite seleccionar ataques a través de una lista similar al diseño clásico del juego. Al seleccionar un movimiento, la aplicación ejecuta la lógica correspondiente y actualiza la interfaz sin interrupciones.

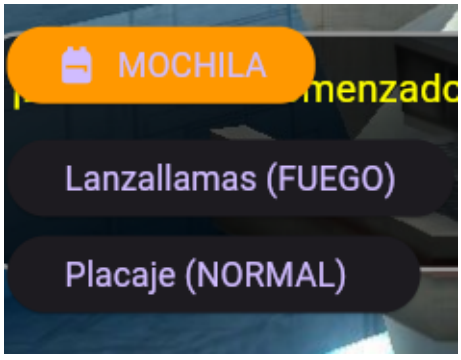


Figura 2: Ataques y mochila

-

Uno de los resultados más importantes fue la incorporación de objetos como característica adicional. El sistema permite que el jugador utilice objetos curativos durante su turno para restaurar parte de la vida del Pokémon. Esta funcionalidad fue implementada y se verificó que el uso de objetos no permite sobrepasar el valor máximo de HP determinado para cada criatura. Asimismo, la aplicación alterna adecuadamente entre atacar y usar objetos según la opción seleccionada por el usuario.

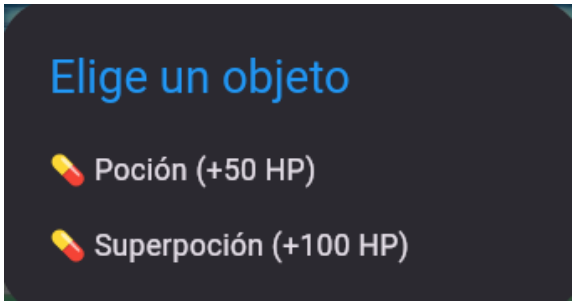


Figura 3: Objetos de la mochila

Para asegurar el correcto funcionamiento de la lógica, se realizaron pruebas mediante el archivo *prueba_logica.dart*. En estas pruebas se confirmó que los cálculos de daño, el orden de turnos, la aplicación de objetos y la detección del final de la batalla operan sin anomalías. La batalla concluye en cuanto uno de los Pokémon llega a cero puntos de vida y la aplicación muestra el resultado final del combate.

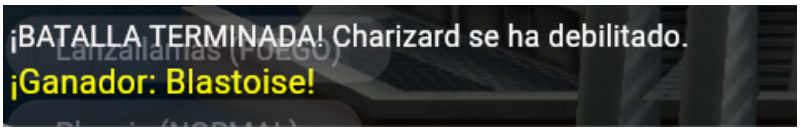


Figura 4: Batalla terminada

Estos resultados muestran que la aplicación cumple con los requerimientos principales del proyecto: ofrece un sistema de combates por turnos, respeta las relaciones de tipos y añade profundidad estratégica mediante el uso de objetos. Las evidencias obtenidas mediante las capturas del funcionamiento demuestran que la aplicación responde adecuadamente y permite desarrollar batallas completas de manera intuitiva.

5. Conclusión

El desarrollo de esta aplicación nos permitió integrar de manera práctica los conceptos fundamentales revisados durante el curso, particularmente aquellos relacionados con la programación orientada a objetos, la lógica estructurada y el diseño de interfaces en Flutter. A través de la implementación del sistema de batalla, se comprobó que es posible modelar comportamientos complejos como turnos, ataques, relaciones de tipos y variación de estadísticas utilizando clases, métodos y estructuras bien definidas. Esto demuestra el valor de las bases teóricas del curso al aplicarse directamente en el diseño de un sistema interactivo.

La simulación de batallas Pokémon desarrollada cumplió con los requisitos planteados inicialmente: se logró integrar un sistema de turnos basado en la velocidad de los Pokémon, una lógica de daño que respeta ventajas y desventajas de tipos, y un menú de acciones que permite ejecutar ataques o utilizar objetos estratégicamente. La correcta actualización de la interfaz y la estabilidad de la aplicación durante las pruebas muestran que Flutter es una herramienta adecuada para construir proyectos interactivos y visualmente claros.

La inclusión de objetos como funcionalidad adicional enriqueció la dinámica del combate, permitiendo al usuario tomar decisiones más variadas y agregando un elemento táctico al flujo del juego. Este aspecto permitió explorar temas adicionales como el manejo de estados internos, el control del flujo del combate y la vinculación entre interfaz y lógica de programación.

En general, el proyecto permitió reforzar habilidades de diseño, análisis y desarrollo de software, además de fomentar la creatividad al adaptar las mecánicas del videojuego original a un entorno propio.

Referencias

[1] (2025) Dart language. Dart Developers. Consultado el 7 de noviembre de 2025. [Online]. Available: https://dart.dev/language

[2] D. T. . D. Programming. (2025) Generic in dart. Consultado el 7 de noviembre de 2025. [Online]. Available: https://dart-tutorial.com/object-oriented-programming/generics-in-dart/

[3] Jaimetellezb. (2025) Dart enums. Medium. Consultado el 23 de noviembre de 2025. [Online]. Available: https://medium.com/@jaimetellezb/dart-enums-2e7c283715a5

[4] (2025) Colecciones en dart. Dart Developers. Consultado el 23 de noviembre de 2025. [Online]. Available: https://dart.dev/language/collections

[5] J. T. B. (2025) Flutter widgets básicos. Consultado el 26 de noviembre de 2025. [Online]. Available: